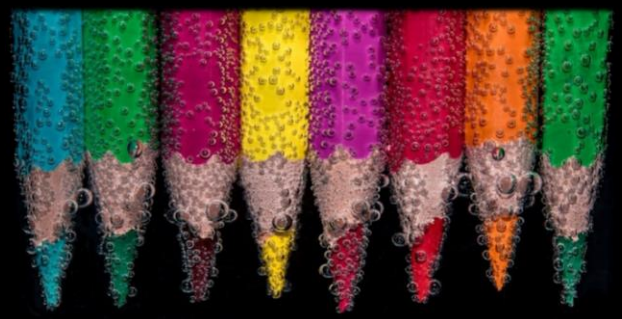
Generics

Adding Type Safety and Code Reusability

SoftUni Foundation



SoftUni Team

Technical Trainers

Software University

http://softuni.bg

Table of Contents

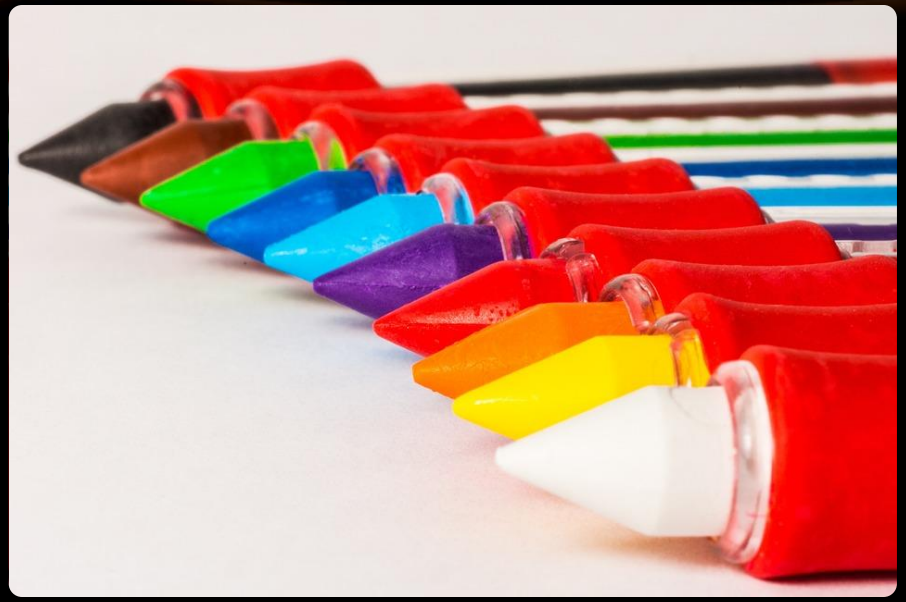
Questions

SoftUni Foundation

sli.do

#JavaFundamentals

SoftUni Foundation



Generics

The Problem, The Solution

The Problem before Java 5.0

SoftUni Foundation

- We need a collection that will store **only strings**

```
List strings = new ArrayList();
strings.add("1");
strings.add("2");
strings.add(3); //Is this correct?
String e1 = (String) strings.get(0);
String e2 = (String) strings.get(1);
String e3 = (String) strings.get(2); // RTE
```

Generics – Type Safety

SoftUni Foundation

- We need a collection that will store **only strings**

```
List<String> strings = new ArrayList<String>();
strings.add("1");
strings.add("2");
strings.add(3); // Compile time error
```

- Adds **type safety** and provides powerful way for **code reuse**

```
List<Integer> strings = new ArrayList<>();
List<Person> people = new ArrayList<>();
```

Type Inference

Generic Classes

- Defined with <**Type Parameter 1**, **Type Parameter 2** … etc.>

```
class ArrayList<T> {
    /* voodoo magic */
}
```

- **Multiple** Type Parameters

```
class HashMap<K, V> {
    /* voodoo magic */
}
```

Type Parameter Scope

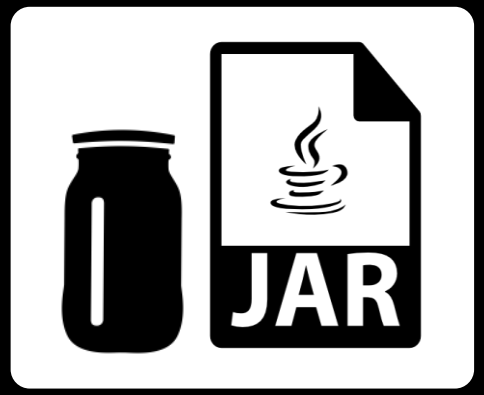
SoftUni Foundation

- You can use it anywhere inside the **declaring class**

```
class List<T> {
    public add (T element) {…}
    public T remove () {…}
    public T get(int index) {…}
}
```

Problem: Jar of T

- Create a class `Jar<>` that can store **anything**
- Adding should add **on top** of its contents
- Remove should get the **topmost** element
- It should have two public methods:
  - `void add(element)`
  - `element remove()`



Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Solution: Jar of T

SoftUni Foundation

```
public class Jar<T> {
    private Deque<T> content;
    public Jar() { this.content = new ArrayDeque<>(); }
    public void add(T entity) {
        this.content.push(entity);
    }
    public T remove() { return this.content.pop(); }
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Subclassing Generic Classes

SoftUni Foundation

- Can **extend** to a concrete class

```
class JarOfPickles extends Jar<Pickle> {
    …
}
```

```
JarOfPickles jar = new JarOfPickles();
jar.add(new Pickle());
jar.add(new Vegetable()); // Error
```

Generic Interfaces

SoftUni Foundation

- **Generic interfaces** are similar to **generic classes**

```
interface List<T> {
    void add (T element);
    T get (int index);
    …
}
```

```
class MyList implements List<MyClass> {…}
```

```
class MyList<T> implements List<T> {…}
```

Generic Methods

SoftUni Foundation

- Can take **generic input** and return **generic output**

between modifiers and return type

```
static <T> List<T> createList(T item, int count) {
  List<T> list = new ArrayList<>();
  for (int i = 0; i < count; i++) {
    list.add(item);
  }
  return list;
}
```

Problem: Generic Array Creator

SoftUni Foundation

- Create a class **`ArrayCreator`** with a single method:
  - **`static T[] create(int length, T item)`**
- Add a single overload:
  - **`static T[] create(Class<T>, int length, T item)`**
- It should **return an array**
  - with the given length
  - every element should be **set to the given default item**

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Solution: Generic Array Creator

SoftUni Foundation

```
public static <T> T[] create(int length, T item) {
   T[] array = (T[]) new Object[length];
   for (int i = 0; i < array.length; i++) {
      array[i] = item;
   }
   return array;
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Solution: Generic Array Creator (2)

SoftUni Foundation

```
public static <T> T[] create(
        Class<T> cl, int length, T item) {
  T[] array = (T[]) Array.newInstance(cl, length);
  for (int i = 0; i < array.length; i++) {
    array[i] = item;
  }
  return array;
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Type Erasure

SoftUni Foundation

- Generics are **compile time illusion**

```
List<String> strings = new ArrayList<String>();
System.out.println(strings instanceof List);


System.out.println(
  strings instanceof List<String>); // CTE
```

- Compiler **deletes** all angle bracket syntax
- Adds **type casts** for us (presented in byte-code)

Type Erasure – Example

SoftUni Foundation

```
public class Illusion<T> {


    public void function(Object obj) {
        if (obj instanceof T) {} // Error
        T[] array = new T[1]; // Error
        T newInstance = new T(); // Error
        Class cl = T.class; // Error
    }
}
```

SoftUni Foundation

Working with Generics

Live Exercises in Class (Lab)

SoftUni Foundation

Type Parameter Bounds

Upper and Lower Bounds

Type Parameter Bounds

SoftUni Foundation

- `<T extends Class>` - specifies an **"Upper bound"**

```
class AnimalList<T extends Animal> {
  private List<T> animals;
  void add (T animal) {…}
  void putAnimalsToSleep() {
    for (Animal a : this.animals)
      a.sleep();
  }}
```

T will be a subclass of Animal

We can now use methods of T

Problem: Generic Scale

SoftUni Foundation

- Create a class `Scale<T>` that:
  - Holds two elements: **left** and **right**
  - Receives the elements through its single constructor:
    - Scale(`T` left, `T` right)
  - Has a method: `T getHeavier()`
- The greater of the two elements is heavier
- Should return `null` if the elements are equal

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Solution: Generic Scale

SoftUni Foundation

```
public class Scale<T extends Comparable<T>> {
    private T left;
    private T right;
    public Scale(T left, T right) {
        this.left = left;
        this.right = right;
    }
    public T getHeavier() { /* next slide */ }     }
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Solution: Generic Scale (2)

SoftUni Foundation

```
public T getHeavier() {
  if (left.compareTo(right) == 0) {
    return null;
  }
  if (left.compareTo(right) < 0) {
    return right;
  }
  return left; }
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Problem: List Utilities

- Create a class `ListUtils` that:
  - Has two static methods:
    - `T getMin(List<T> list)`
    - `T getMax(List<T> list)`
  - Should throw `IllegalArgumentException` if an empty list is passed

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Solution: List Utilities

SoftUni Foundation

```
public static <T extends Comparable<T>> T getMax(List<T> list) {
  if (list.size() == 0) throw new IllegalArgumentException();
  T max = list.get(0);
  for (int i = 1; i < list.size(); i++) {
    if (max.compareTo(list.get(i)) < 0) {
      max = list.get(i);
    }
  }
  return max;  }
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Type Parameters Relationships

SoftUni Foundation

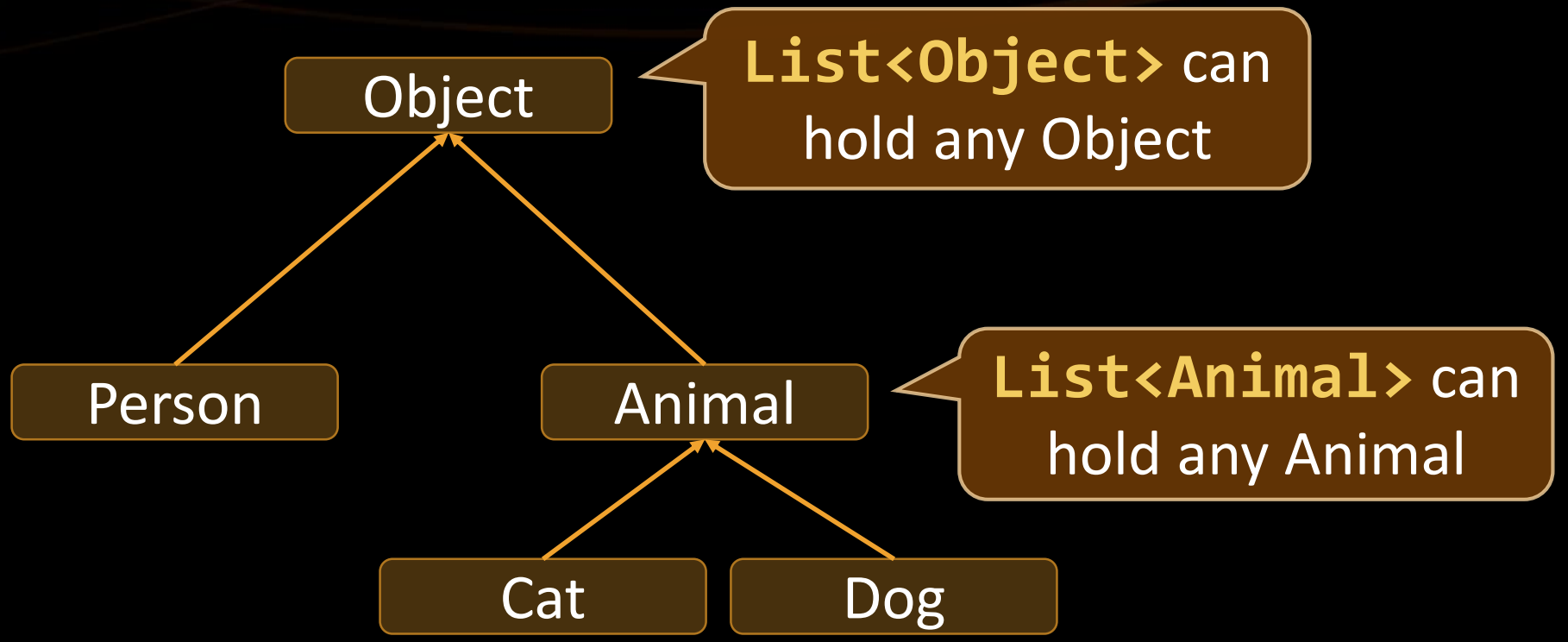
- Generics are **invariant**

```
List<Object> objects = new ArrayList<>();
List<Animal> animals = new ArrayList<>();
objects = animals; // Compile Time Error!
```

- If the above was possible, then why not:

```
objects = animals;
objects.add(new Person()); // Impossible!
```

Type Parameters Relationships (2)




```
List<Object> ≠ List<Animal>
```

Wildcards

- Wildcards introduce **polymorphism to type parameters**

```
List<Number> numbers = new ArrayList<>();
List<Integer> integers = new ArrayList<>();



// The Problem
numbers = integers; // Compile Time Error!
```

We can fix this using wildcards

Unbounded Wildcards

SoftUni Foundation

- <?> - specifies a **Type** that can be any **Type** (i.e. **extends Object**)

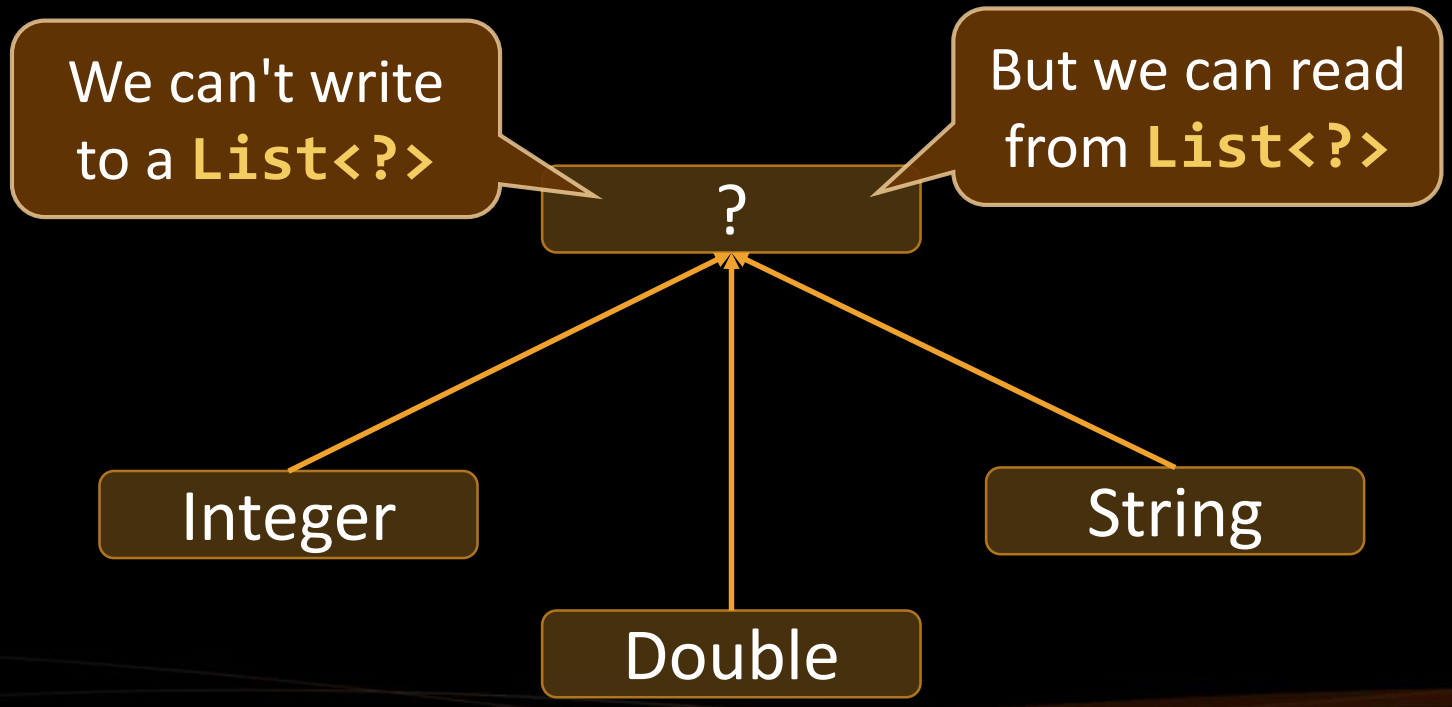
```
List<?> anyList;


List<Integer> integers = new ArrayList<>();
List<Double> doubles = new ArrayList<>();
anyList = integers; // OK
anyList = doubles; // OK
anyList.add(1); // NOT OK!
```

Unknown type parameter!

Unbounded Wildcards (2)

- `List<?>` can be a `List<Integer>`
- `List<?>` can also be a `List<Double>`
- `List<?>` can be of any Type (`<? extends Object>`)



Problem: Null Finder

- Add a method to your **List Utilities** class that finds the index of every `null` element in a given list:
  - `static List<Integer> getNullIndices(List<> list)`
- Add the appropriate generic syntax to the signature
- The method should work with any `List<>`

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Solution: Null Finder

SoftUni Foundation

```
public static Iterable<Integer> getNullIndices(List<?> list) {
  Collection<Integer> nulls = new ArrayList<>();
  for (int i = 0; i < list.size(); i++) {
    if (list.get(i) == null) {
      nulls.add(i);
    }
  }
  return nulls; }
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Bounded Wildcards – Upper Bounds

SoftUni Foundation

- `<? extends Number>` - subtype of **Number**

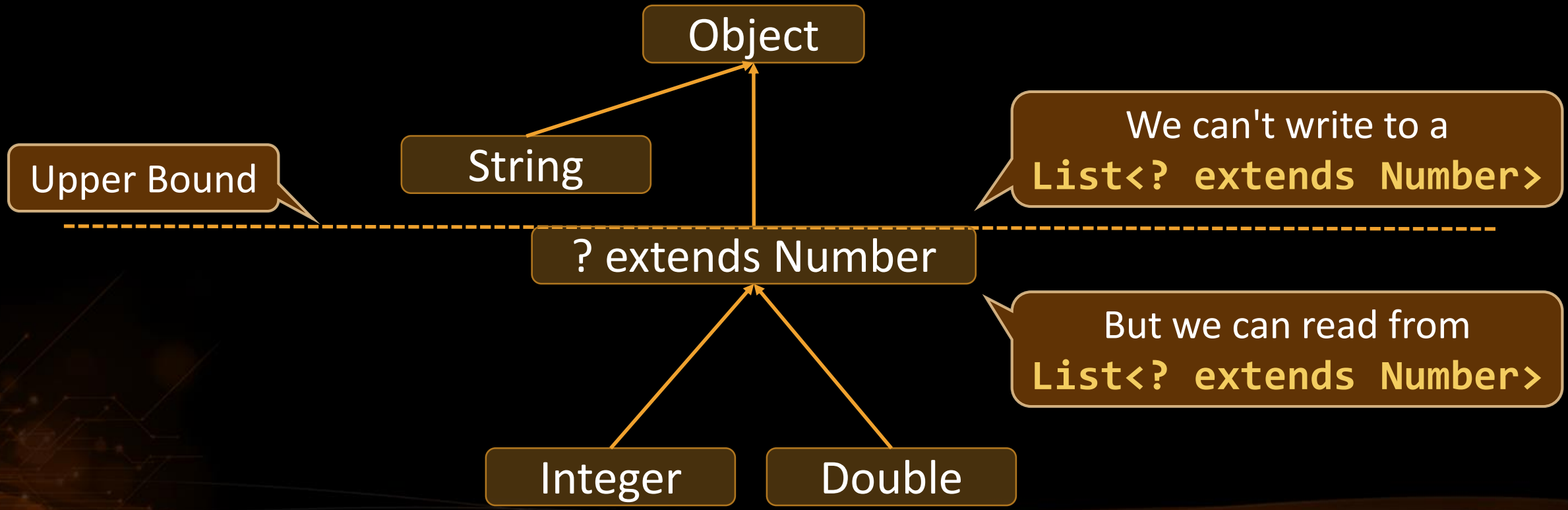
```
List<? extends Number> numbers;


List<Integer> integers = new ArrayList<>();
List<Double> doubles = new ArrayList<>();
numbers = integers; // OK
numbers = doubles; // OK
numbers.add(1); // NOT OK!
```

Bounded Wildcards – Upper Bounds (2)

SoftUni Foundation

- `List<? extends Number>` can be a `List<Integer>`
- `List<? extends Number>` can also be a `List<Double>`



Bounded Wildcards – Lower Bounds

SoftUni Foundation

- `<? super Class>` - Any supertype of Class (i.e. **super Class**)

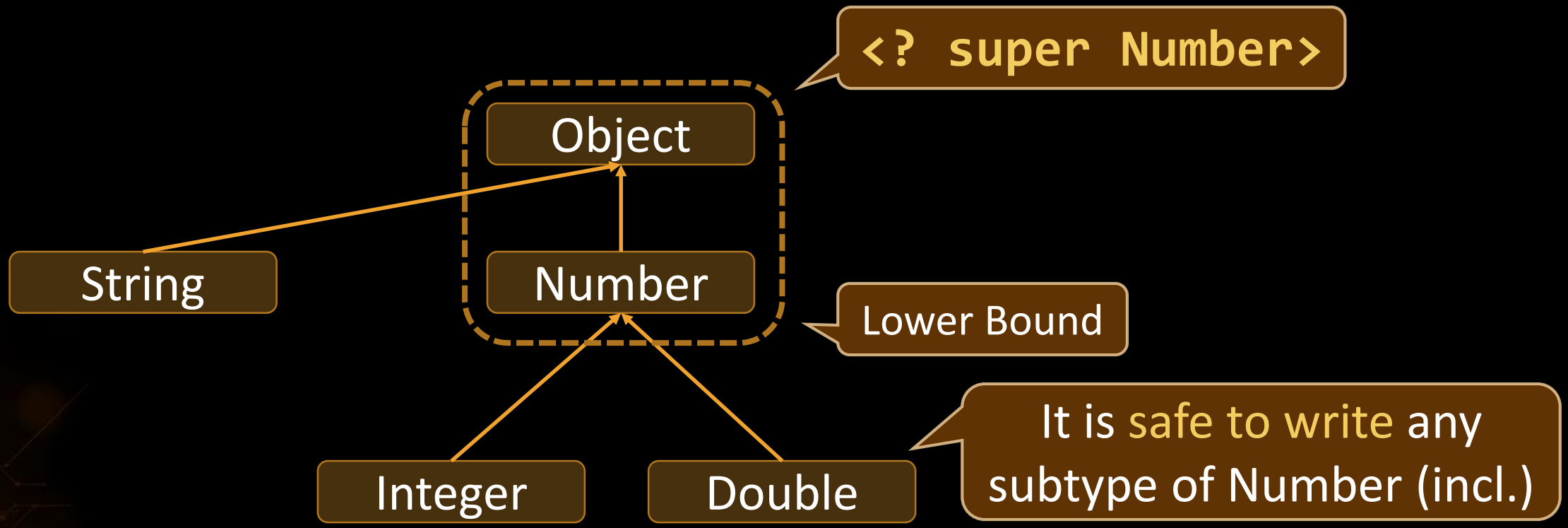
```
List<? super Number> super;


List<Integer> integers = new ArrayList<>();
List<Object> objects = new ArrayList<>();
super = objects; // OK!
super.add(1); // OK!
super = integers; // NOT OK!
```

Bounded Wildcards – Lower Bounds (2)

SoftUni Foundation

- `List<? super Number>` can be a `List<Number>`
- `List<? super Number>` can also be a `List<Objects>`



Problem: Generic Flat Method

SoftUni Foundation

- In `ListUtils`, create a generic static method that flattens a `List<List<>>` into a resulting `List<>`
- Signature:
  - `void flatten(List<> dest, List<List<>> src)`
- Add the appropriate generic syntax to the signature
- The method should work with any `List<>`

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Solution: Generic Flat Method

SoftUni Foundation

```
public static <T> void flatten(
        List<? super T> dest, List<List<? extends T>> src) {


  for (List<? extends T> inner : src) {
    dest.addAll(inner);
  }
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Problem: Generic Add All

- In `ListUtils`, create a generic static method that adds all elements from a **given source** list to a **given destination** list
  - `void addAll(List<> destination, List<> source)`

- Add the appropriate generic syntax to the signature
- The method should work with any `List<>`

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

Solution: Generic Add All

SoftUni Foundation

```
public static <T> void addAll(
        List<? super T> dest, List<? extends T> source) {


  for (T element : source) {
    dest.add(element);
  }
}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/521#0

SoftUni Foundation

Working with Generic Bounds

Live Exercises in Class (Lab)

Summary

- Generics add **type safety**
- Generic code is more **reusable**
- **Classes**, **interfaces** and **methods** can be generic
- Runtime information about **type parameters** is lost due to **erasure**
- **Wildcards** introduce **polymorphism** to type parameters
- Type parameters can have **lower** or **upper bounds**

Generics

Questions?

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University Foundation
  - http://softuni.foundation/
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg

Software University

SoftUni Foundation

FORUM

License

SoftUni Foundation

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

- Attribution: this work may contain portions from
  - "Fundamentals of Computer Programming with Java" book by Svetlin Nakov & Co. under CC-BY-SA license
  - "C# Part I" course by Telerik Academy under CC-BY-NC-SA license
  - "C# Part II" course by Telerik Academy under CC-BY-NC-SA license